Efficient and Transparent Instrumentation using Dynamic Compilation

Robert Cohn, P Geoffrey Lowney, CK Luk,
Robert Muth, Harish Patil
Intel
Hudson, MA

Instrumentation

```
Max = 0;
for (p = head; p; p = p->next)
{
   printf("Loop\n");
   if (p->value > max)
   {
      printf("max\n");
      max = p->value;
   }
}
```

Dynamically defined

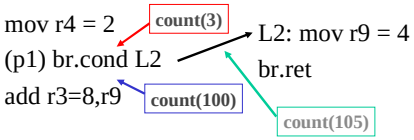
Pin: Tool for Making Tools

- Profiler for compiler optimization:
  - Basic-block count
  - Value profile
- Micro architectural study:
  - Instrument branches to simulate branch predictors
  - PinPoints (Pin + SimPoint)
  - Fault Insertion (Princeton)
- Bug checking:
  - Find references to uninitialized, unallocated data
- Software tools that use instrumentation:
  - Purify, Valgrind, Vtune
- Platforms, Linux IA32, Itanium, ARM

Pin Provides Low Level View of Program Execution

Insert Calls Relative to an instruction:

1. Before
2. After
3. Taken edge of branch



Example: Instruction Trace

```
[rscohn1@shli0005 Trace]$ ./hello
Hello world
[rscohn1@shli0005 Trace]$ itrace -- ./hello
Hello world
[rscohn1@shli0005 Trace]$ head prog.trace
0x20000000000045c0
0x20000000000045c1
0x20000000000045c2
0x20000000000045d0
0x20000000000045d2
0x20000000000045e0
[rscohn1@shli0005 Trace]$
```

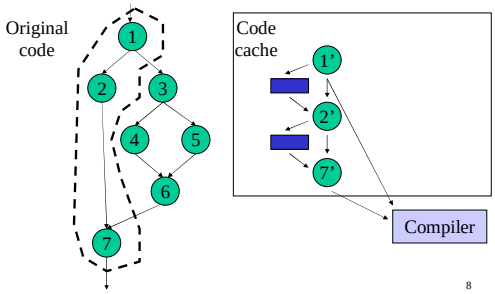

Example: Instruction Trace

traceInst(ip);
mov r2 = 2
traceInst(ip);
add r3 = 4, r3
traceInst(ip);
(p2) br.cond L1
traceInst(ip);
add r4 = 8, r4
traceInst(ip);
br.cond L2

```
#include <stdio.h>
#include "pinstr.H"
FILE *traceFile;
void traceInst(long * ipsyll){
    fprintf(traceFile, "%p\n", ipsyll);
}
void Instruction(INS ins, void *v){
    PIN_InsertCall(IPOINT_BEFORE, ins,
   (AFUNPTR)traceInst, IARG_IP_SLOT, IARG_END);
}
int main(int argc, char *argv[])
{
    PIN_AddInstrumentInstructionFunction(Instruction,0);
    traceFile = fopen("prog.trace", "w");
    PIN_StartProgram();
}
```

7

Execution Drives Instrumentation



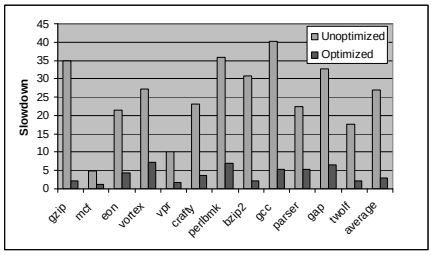
8

Why Dynamic Code Generation?

- Dynamic instrumentation
- More transparent to user
  - Shared libraries, dynamically generated code are instrumented
  - Handles variable size instructions, data in text section
  - No special compiler/linker switches
- Makes simple user model efficient
  - Specialization enables optimization

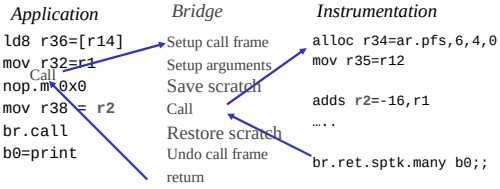
9

Overhead of Instrumentation Basic Block Counting



10

Where Does the Overhead Come From?



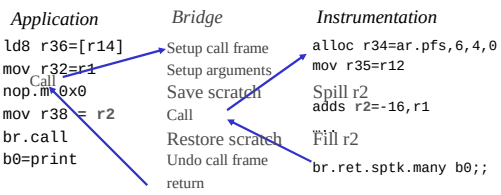
11

Instrumentation Overhead

- Itanium has much more architectural state than other processors
  - Many scratch registers: 120
  - Data/Control Speculation
  - Register Stack Engine
  - Rotating register files
- Simple and flexible user model has a high cost
- Pin uses global optimization to reduce the cost

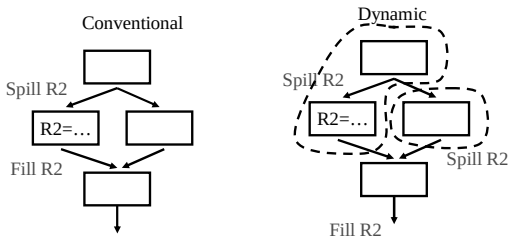
12

Calling Instrumentation Routine

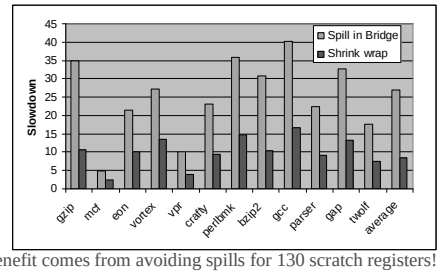


Dynamic shrink wrapping:
Defer spilling application registers until first use

Dynamic Shrink Wrapping



Shrink Wrapping

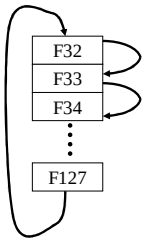


Benefit comes from avoiding spills for 130 scratch registers!
Optimal spill placement is less important

Register Spill Problems

- no compiler cooperation
  - Rotating register files
  - ALAT for data speculation
  - NAT bits for control speculation

Spilling Rotating Registers



Loop branches cause register rotation

Spilling Rotating Registers

```
L1:
f40 = f50 + f21
(p17) br.wtop L1
```

- After a rotation, which registers are spilled/need to be spilled?

Spilling Rotating Registers

- Before first rotate, spill all unspilled application rotating fp registers

```
L1:
spill f40
f40 = f41 + f21
spill f32-39,42-f127
(p17) br.wtop L1
```

Specialized:
Rotating registers are spilled

```
L1:
f40 = f41 + f21
(p17) br.wtop L1
```

FP register spill is usually outside loop

19

Spilling ALAT Slots

- Data speculation: compiler moves a load above a store but may be to same address
- Addresses of data speculative loads are stored in ALAT, indexed by register
- Mis-speculated loads are re-executed

```
ld8.a r2 = [100200]
st8 [r4] = r3
chk.a.clr r2, recovery01
```

ALAT
1:
2: 100200
3:

20

Spilling ALAT Slots

- Filling to a register with stale ALAT data can cause error
- It is always correct to clear ALAT – makes future checks fail
- Track if ALAT entry is known clear
- Avoid spilling registers when ALAT may not be clear
- If ALAT entry is unknown, clear it on a fill
  - Specialization

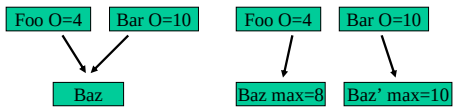
21

Register Stack Engine

- Instrumentation needs new frame on register stack
- Pin must select frame size in alloc instruction big enough to cover existing frame
- Usually 8, but could be 96
- Conservatively large register frame causes extra memory traffic

22

Bounding Output Frame Size

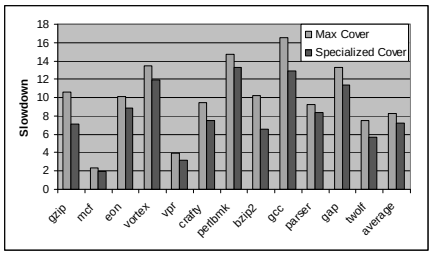


Output frame size comes from caller

Specialize Baz for max = 8
Specialize Baz` for max=10

23

Register Stack Engine Optimization



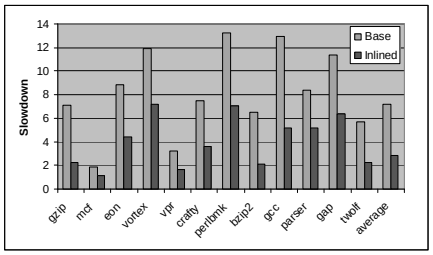
24

Inlining

- Simple instrumentation routines are inlined into application code

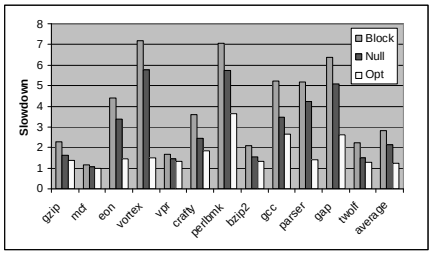
25

Inlining



26

How Much Opportunity is Left?



27

Summary

- Download a kit: http://rogue.colorado.edu/Pin
- Pin is an easy to use and flexible tool for instrumentation of Itanium/IA32/ARM/Linux programs
- Large architectural state makes efficient instrumentation more challenging
- Shrink wrapping, inlining, and specialization are effective in reducing the overhead of instrumentation (27x $\Rightarrow$ 2.8x)

28